Internal Strategies in a Rewriting Implementation of Tile Systems [1]

R. Bruni [a], J. Meseguer [b], and U. Montanari [a]

[a] *Dip. di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy*

[b] *CSL, SRI International, 333 Ravenswood Ave. Menlo Park, CA 94025, USA*

Abstract

Tile logic extends *rewriting logic*, taking into account rewriting with side-effects and rewriting synchronization. Since rewriting logic is the semantic basis of several language implementation efforts, it is interesting to map tile logic back into rewriting logic in a conservative way, to obtain executable specifications of tile systems. The resulting implementation requires a meta-layer to control the rewritings, so that only tile proofs are accepted. However, by exploiting the *reflective* capabilities of the *Maude* language, such meta-layer can be specified as a kernel of *internal strategies*. It turns out that the required strategies are very general and can be reformulated in terms of search algorithms for non-confluent systems equipped with a notion of success. We formalize such strategies, giving their detailed description in Maude, and showing their application to modeling *uniform* tile systems.

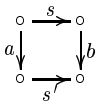
1 Introduction

The evolution of a process in a concurrent system often depends on the behaviours of other cooperating processes. For example, in some critical states, a process must have the opportunity to check incoming communications from many sources, without granting a privilege to some source or to a particular kind of input. Thus, a specification language for concurrent systems

[1] Research supported by Office of Naval Research Contracts N00014-95-C-0225 and N00014-96-C-0114, by National Science Foundation Grant CCR-9633363, and by the Information Technology Promotion Agency, Japan, as part of the Industrial Science and Technology Frontier Program "New Models for Software Architechture" sponsored by NEDO (New Energy and Industrial Technology Development Organization). Also research supported in part by U.S. Army contract DABT63-96-C-0096 (DARPA); CNR Integrated Project *Metodi e Strumenti per la Progettazione e la Verifica di Sistemi Eterogenei Connessi mediante Reti di Comunicazione*; and Esprit Working Groups *CONFER2* and *COORDINA*. Research carried out in part while the first and the third authors were visiting at Computer Science Laboratory, SRI International, and the third author was visiting scholar at Stanford University.

cannot leave out of consideration some mechanism for expressing (guarded) non-deterministic choices in the body of a process. Such a mechanism should allow local choices to be coordinated, affecting the global evolution of the system.

The *tile model* [17,19] is a formalism for modular descriptions of concurrent systems. Basically, a set of rules defines the behaviour of certain modules (a module is just an open, e.g., partially specified, configuration of the system), which may interact through their interfaces. Then, the behaviour of a system as a whole consists of a coordinated evolution of its sub-modules. Each rule has the form:



also written $s \xrightarrow[b]{a} s'$, stating that the *initial configuration* s of the system can evolve to the *final configuration* s' producing an *effect* b, but the step is allowed only if the subcomponents of s evolve to the subcomponents of s', producing the *trigger* a. The vertices $\circ$ of the tile are called *interfaces*. More complex rules can be generated by composing tiles horizontally (through side effects), vertically (building conditional computations of a certain component), and in parallel (concurrent steps).

By analogy with *rewriting logic* [24], where a logic theory is associated to a term rewriting system in such a way that each computation represents a *sequent* entailed by the theory, the tile model also comes equipped with a purely logical presentation [19], where tiles are just considered as special sequents subject to certain inference rules. The *entailment* relation is specified by simple inference rules, and equivalent computations yield exactly the same sequent. In this sense, *tile logic* is a logic of concurrent systems with synchronization mechanisms.

Tile logic extends rewriting logic (in the non-conditional case), taking into account rewriting with side effects and rewriting synchronization. On the other hand, since there exist several languages based on rewriting logic (Cafe [16], ELAN [3], Maude [9]), the implementation of a conservative mapping of tile logic into rewriting logic would facilitate the execution and development of tile specifications. This topic has been extensively investigated in [26,6]. From a practical point of view, the mapping becomes effective provided that the rewriting engine is able to select, among all the possible rewriting computations, those interpreting tile logic derivations. For this purpose, we exploit the *reflective* capabilities [10,11] of the Maude language [9,8] developed at SRI, defining suitable *internal strategies* [12], which can help the user to collect and analyze the possible computations and results. A key point is that the internal strategies needed to embed tile systems in rewriting logic are for the most part general meta-strategies for nondeterministic rewriting systems. We give a precise description of such strategies, and of their application to the

implementation of a large class of tile systems (called *uniform* tile systems).

The structure of the paper is as follows. In Section 2 we give a survey of tile logic and its translation into rewriting logic, recalling the results from [26,6]. In Section 3 we address the issue of non-confluent rewrite systems, and propose a meta-layer of internal strategies, written in a self-explanatory Maude-like notation, for collecting results and embedding tile systems, and in Section 4 we formalize their application to uniform tile systems, and in particular to the simple yet interesting example of finite CCS.

The definition of internal strategies to control nondeterministic rewritings in the tile system translations constitutes the main contribution of this paper. The importance of similar mechanisms is well-known, and other languages (e.g., ELAN), have built-in constructs to deal with general forms of nondeterminism. Nevertheless, our approach is rather general (it is parametric w.r.t. a user-definable success predicate) and allows the application of several visiting policies, different from the depth-first (with backtracking) algorithms that are usually preferred in a built-in implementation for efficiency reasons, but that could diverge also in the presence of solutions.

This work is part of our ongoing research on general mechanisms for the rewriting implementation of interesting classes of tile systems. An extensive presentation of this topic can be found in [6] (other references are [26] and the forthcoming PhD thesis of one of the authors [5]).

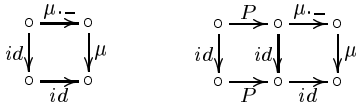
2 Mapping Tile Logic into Rewriting Logic

The notions of *configuration* and *effect* come naturally equipped with operations of parallel and sequential composition. In particular they form two *monoidal categories* having the same class of objects. The use of categories offers a convenient characterization of configurations and effects also in terms of *algebraic theories* [21]. The free algebraic theory associated to a (one-sorted) signature Σ is called the *Lawvere theory* for Σ, and is denoted by $\mathcal{L}_\Sigma$: the objects are underlined natural numbers, the arrows from $\underline{m}$ to $\underline{n}$ are in a one-to-one correspondence with n-tuples of terms of the free Σ-algebra with (at most) m variables, and composition is term substitution. It has been shown in [24], that a rewriting theory $\mathcal{R}$ yields a cartesian *2-category*[2] $\mathcal{L}_\mathcal{R}$, which does for $\mathcal{R}$ what a Lawvere theory does for a signature. Gadducci and Montanari pointed out in [18] that if side-effects are also introduced, then *double categories* [14,1,20] should be considered in place of 2-categories. A double category can be informally described as the superposition of a horizontal and a vertical category of *cells*, the former defining effect propagations, and the latter describing state evolutions. Then, in the same way as the term algebra is freely generated by a signature, and the initial model of rewriting logic is

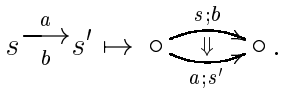
[2] A 2-category [20,22] is a category $\mathcal{C}$ such that, for any two objects a, b, the class $\mathcal{C}[a,b]$ of arrows from a to b in $\mathcal{C}$, also forms a category and satisfies particular composition properties.

freely generated from the rules of the rewriting system, the tiles freely generate a double category which gives the natural operational characterization[3] of the system, in the spirit of initial model semantics.

Though we do not want to stress here the elegance, expressiveness and advantages of tile logic as a computational paradigm, a simple example is necessary to show how tiles can help in formally reasoning about process algebras. Let us consider the usual *action prefix* operation, denoted by $\mu.\_$. The corresponding tile is represented below and can be composed horizontally with the identity cell of any process P to model the transition $\mu.P \xrightarrow{\mu} P$ associated to the action prefix.



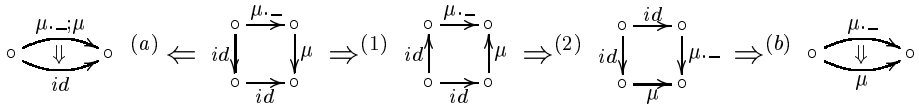
Let nil be the inactive process, and consider the process $p = \mu_1.\mu_2.nil$. If the process p tries to execute μ_2 before executing μ_1 it gets stuck, because there is no tile having μ_2 as trigger and $\mu_1.\_$ as initial configuration. In a non-conditional rewriting system, this is not necessarily true, because rewriting steps can be freely contextualized (and instantiated). This problem is well-known, and some ad-hoc solutions have been already proposed in the literature [23,28]. Our methodology offers a unifying view for many analogous situations. The basic idea is to "stretch" tiles into ordinary rewriting rules, preserving the capacity to distinguish between configurations and effects:



In [26,6], the comparison between tile logic and rewriting logic takes place by embedding their categorical models in a recently developed, more general framework, called *partial membership equational logic* (PMEqtl) [25]. Indeed, using PMEqtl, it is possible to define an extended version of 2-categories, where the distinction between effects and configurations can be expressed through membership predicates. PMEqtl is particularly suitable for the embedding of categorical structures, firstly because the sequential composition of arrows is a partial operation, and secondly because membership predicates over a poset of sorts allow the objects to be modelled as a subset of the arrows and arrows as a subset of cells. Moreover, the *tensor product construction* illustrated in [26] can be easily expressed in PMEqtl, yielding a convenient formulation of monoidal double categories. As a main result, given a tile system $\mathcal{R}$, a sequent $s \xrightarrow[b]{a} s'$ is entailed by $\mathcal{R}$ in tile logic (written $\mathcal{R} \vdash s \xrightarrow[b]{a} s'$) if and only if the sequent $s;b \Rightarrow a;s'$ is entailed by the stretched version of $\mathcal{R}$ in rewriting logic and its proof satisfies some additional constraints (see [6]).

[3] Tiles are double cells, configurations are horizontal arrows, effects are vertical arrows, and objects model connections between the somehow syntactic horizontal category and the dynamic vertical evolution.

Moreover, for a large class of tile systems (called *uniform*) the additional constraints can be verified just by inspecting the border of the sequent. It follows that a typical query in a tile system is: "derive all (some of) the tiles with a given horizontal source s and vertical target b". A surprising feature in the translation of a tile system is that queries start with a vertical target rather than with a source. If the vertical arrows are terms, then this is the only correct procedure. However, for CCS-like process algebras, we realized that the vertical and horizontal dimensions can be swapped in such a way that the queries are of the kind "derive all one-step transitions for a given agent P". This is possible because the vertical signature consists of *unary* actions. So we can: (1) reverse the vertical arrows in the tile system and then (2) rotate clockwise the tiles by 90 degrees before their translation in ordinary rewrite rules, as illustrated below for the action prefix tile:



Let us examine the different 2-cell translations. The cell (a) on the left states that if we force the process $\mu.P$ to perform a μ action, it succeeds. The other cell (b) states that process $\mu.P$ may perform the action μ. Therefore, an implementation using (a)-rules can only *test* CCS processes, whereas using (b)-rules, all the possible behaviours of a CCS process can be collected. In both cases we must explore the tree of nondeterministic rewritings until a correct final configuration is reached.

3 Dealing with Nondeterminism

The theoretical results summarized in the previous section, cannot be applied to get an immediate rewriting implementation of tile systems, because of the additional constraints that the proofs of the rewriting computations must satisfy. Indeed, a tile computation coordinates the activities of each module, whereas the coordination layer is missing in the stretched version. In particular, the absence of side effects allows each module to evolve separately, but if the local choices are not correct, their synchronization could become unfeasible. Therefore, we need a methodological approach to drive computations along correct paths. In this section we illustrate the more general problems arising in a non-confluent rewrite system, and a solution based on internal strategies in reflective languages.

3.1 Nondeterministic Rewriting Systems

In rewriting logic, nondeterminism can arise whenever multiple rewritings are enabled for the same term. For example the conditional rewriting rules

```
crl t(x⃗) => t1(x⃗) if G1(x⃗) .
```
⋮

`crl t(`$\vec{x}$`) => t`$_n$`(`$\vec{x}$`) if G`$_n$`(`$\vec{x}$`) .`

describe a system in which the terms matching `t(`$\vec{x}$`)` can be nondeterministically rewritten into n different terms (in what follows the conditions `G`$_i$`(`$\vec{x}$`)` are called *guards*, and we say that a guard is *satisfied* if it is evaluated to `true` and that it *fails* if it is evaluated to `false`). If several guards among the `G`$_i$`(`$\vec{x}$`)` are satisfied, then the rule to be applied is chosen by the rewrite engine accordingly to some general policy. We can distinguish between three nondeterministic mechanisms, namely *conditional choice*, *don't know nondeterminism*, and *don't care nondeterminism*.

- **Conditional choice**: the guards are sequentially processed according to their listing order. The first satisfied guard, say `G`$_i$`(`$\vec{x}$`)`, selects its corresponding i-th rule for the rewriting. Using conditional choice the programmer knows which rule will be chosen if more than one is satisfied. On the other hand, due to the explicit *priority* ordering on the clauses, the use of conditional choice can result into a non-fair policy.
- **Don't care (dc) nondeterminism**: in this case, if any (not necessarily the first) of the guards is satisfied, then the corresponding rule can be applied. Here the main assumption is that whatever choice will be selected, the system will continue to behave correctly. For instance, this approach is well suited whenever the Church-Rosser property holds. At the semantic level, dc nondeterminism can overcome the drawback of conditional choice, but the programmer has less control over the computation flow.
- **Don't know (dk) nondeterminism**: sometimes it is not enough to explore just one branch of the nondeterministic computations, because many problems (e.g., in Artificial Intelligence or in Operations Research) are currently solvable only by resorting to some sort of search. In this case the nondeterminism leads to a parallel exploration of the enabled branches. However, performance considerations suggest alternative visiting policies (e.g., depth first with backtracking instead of breadth first). Under some assumption (e.g., finiteness of the tree) the user may explore all branches, and collect all the solutions. According to dk nondeterminism, if only one clause, say `G`$_i$`(`$\vec{x}$`)`, is satisfied, then the rewriting is said to be *determinate* and the i-th rule is applied. If more than one clause are satisfied, then the statement is said to be *nondeterminate* and the alternative computation paths are explored concurrently .

Now let us suppose that the system comes equipped with a general notion of success and failure which is represented by a predicate `ok(_)`, defined over terms. We say that a term t is *final* if `ok(`t`)` $\in$ {`true`,`false`}. Moreover, a computation c of the system is *successful* if c reaches a final term t such that `ok(`t`)` = `true` and for every term t' visited by c `ok(`t'`)` $\neq$ `false`. A computation c is *failing* if `ok(`t'`)` = `false` for some term t' visited by c. Obviously, all the failing computations must be discarded (e.g., as soon as a failure is detected the system stops, and a new run with different choices is

considered); dk nondeterminism allows exploring all the alternatives.

For efficiency reasons, only dc nondeterminism is implemented in Maude's default interpreter. This means that whenever multiple reductions are possible the system arbitrarily executes one of them in a fair top-down fashion, and the user has virtually no control over computations, because the order in which the clauses are listed is not important, or more generally, because although execution paths can be traced, it is difficult to understand how the default strategy determines each choice in a complex example (moreover, in rewriting logic the rules can be applied also to proper subterms, and not only to the "top" of the tree-like structure of terms). However, since Maude is a reflective language, it is possible to overcome this limitation by importing the meta level of some specification, and controlling the computation with suitable (meta-programmed) strategies [12]. We are mostly interested in strategies for dk nondeterminism.

3.2 A Strategy Kernel Language in Maude

Given a logical theory T, a *strategy* is any computational way of looking for certain proofs of some theorems of T. An *internal strategy language* is a theory-transforming function S that sends each theory T to another theory $S(T)$ *in the same logic*, whose deductions simulate controlled deductions of T. Given a logic, we say that it is *reflective*, relatively to a class $\mathcal{C}$ of theories, if we can find inside $\mathcal{C}$ a *universal theory* U where all the other theories in the class $\mathcal{C}$ can be simulated, i.e., there exists a *representation* function

$$\overline{(\_ \vdash \_)} : \bigcup_{T \in \mathcal{C}} \{T\} \times s(T) \longrightarrow s(U)$$

where $s(T)$ denotes the set of meaningful sentences in the language of a theory T, such that for each $T \in \mathcal{C}$ and $\varphi \in s(T)$: $T \vdash \varphi \Longleftrightarrow U \vdash \overline{T \vdash \varphi}$. Therefore, the strategies $S(U)$ for the universal theory are particularly important, since they represent, at the object level, strategies for computing in the universal theory. Moreover, since U itself is representable ($U \in \mathcal{C}$), we get a *reflective tower*: $T \vdash \varphi \Longleftrightarrow U \vdash \overline{T \vdash \varphi} \Longleftrightarrow U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \cdots$

The class of finitely presentable rewrite theories has universal theories, making rewriting logic reflective [10,7]. A rewrite theory T consists of a signature Σ, a set E of equations, and a set of labelled rewrite rules. The deductions of T are rewrites modulo E using such rules, and the meaningful sentences are rewrite sequents $t \Rightarrow t'$, where t and t' are Σ-terms. Let $\mathcal{C}$ be the class of finitely presentable rewrite theories, and let U be a universal theory in $\mathcal{C}$. The representation function $\overline{(\_ \vdash \_)}$ encodes a pair consisting of a rewrite theory T in $\mathcal{C}$ and a sentence $t \Rightarrow t'$ in T as a sentence $\langle \overline{T}, \overline{t} \rangle \Rightarrow \langle \overline{T}, \overline{t'} \rangle$ in U, in such a way that $T \vdash t \Rightarrow t' \Longleftrightarrow U \vdash \langle \overline{T}, \overline{t} \rangle \Rightarrow \langle \overline{T}, \overline{t'} \rangle$, where the function $\overline{(\_)}$ recursively defines the representation of rules, terms, etc. as terms in U.

Maude [9,8] supports an arbitrary number of levels of reflection and gives the user access to important reflective capabilities, including the possibility

of defining and using internal strategy languages, whose correctness relies on a basic *reflective kernel*, that is, on some basic functionality provided by the universal theory U. In particular, the Maude implementation supports metaprogramming of strategies via a built-in module `META-LEVEL`. For example, such a module provides sorts `Term` and `Module`, so that the representations $\overline{t}$ and $\overline{T}$ of a term t and a module T have sorts $\overline{t}$: `Term` and $\overline{T}$: `Module`. Then, the declaration `META-LEVEL[`T`]` imports the module `META-LEVEL`, declares a new constant `T` of sort `Module`, and adds an equation making `T` equal to the representation of T in `META-LEVEL`. Therefore, we can regard `META-LEVEL` as a module-transforming operation that maps a module T to another module `META-LEVEL[`T`]` that is a definitional extension of U. In particular two important operations are defined:

The first is `meta-reduce(`$\overline{T}$`,`$\overline{t}$`)`: it takes the metarepresentations $\overline{T}$ of a module T and $\overline{t}$ of a term t and evaluates as follows: (a) first $\overline{t}$ is converted to the term it represents; (b) then this term is fully reduced using the equations in T; (c) the resulting term t_r is converted to a meta-term which is returned as the result.

The second operation is `meta-apply(`$\overline{T}$`,`$\overline{t}$`,`$\overline{l}$`,`n`)`: it takes the metarepresentations of a module T, of a term t, and of a label l and a natural number n, then it evaluates as follows: (a) first $\overline{t}$ is converted to the term it represents; (b) then this term is fully reduced using the equations in T; (c) the resulting term t_r is matched against all rules with label l, with matches that fail to satisfy the condition of their rule discarded; (d) the first n successful matches are discarded; (e) if there is an $(n+1)$-th match, its rule is applied using that match; otherwise `{error*,empty}` is returned (`empty` represents the empty substitution); (f) if a rule is applied, the resulting term t' is fully reduced using the equations in T; (g) the resulting term t'_r is converted to a meta-term which is returned as a result, paired with the match used in the reduction (the operator `{_,_}` is used to construct the pair).

Remark 3.1 To make easier the notation, we have used a simpler syntax than the one in the Maude implementation, where `meta-apply` has an additional argument representing a substitution σ (possibly empty) for the variables in the rules of T labelled by l, to be applied before the matching with t.

3.3 Collection of rewritings

In this section, we specify a strategy language able to support dk nondeterminism, which consists of a module-transforming operation `ND-SEM` that extends the strategy kernel. In particular, we define three different functionalities whose correctness can be easily derived from the correctness of `meta-apply`.

The first functionality, called `first`, takes as arguments the metarepresentations of a module T, of a term t, of a label l, and a natural number n and evaluates to the sequence of terms containing the first n successful rewritings of t in T using rules labelled by l. If no rewrite is possible, then the empty

list `nilSeq` is returned. If only m rewritings are possible, with $m < n$, then the sequence contains only the corresponding m terms.

A second functionality, called `last`, can collect an unbounded number of possible rewritings. Since the presentation of the theory T is finite and also the term t that one wants to rewrite is a finite term, it follows that there are always a finite number of possible (one step) rewritings for the term t in T. However, it is common that the number of possible rewritings is unknown by the user, so that the `first` operation does not give much help. The function `last` takes as arguments the meta-representations of a module T, of a term t of T and of a label l, and a natural number n. Its evaluation returns the term sequence of all the possible rewritings of t in T, except the first n, using rules with label l. This can be immediately generalized (when $n = 0$) to a function `allRew` taking as arguments the meta-representations of T, t and l and returning all the successful rewritings of t in T using rules with label l. Notice that the specification level is not affected by the meta-extensions. For lack of space, we can present only part of the Maude code.

```
mod ND-SEM is protecting META-LEVEL .
 sort TermSeq . subsort Term < TermSeq .
 op nilSeq : -> TermSeq .
 op seq : TermSeq TermSeq -> TermSeq [assoc id: nilSeq] .
 ops first last : Module Term Label Nat -> TermSeq .
 op allRew : Module Term Label -> TermSeq ...
endm
```

Remark 3.2 The attribute `[assoc]` states that the operator `seq` used to build term sequences is associative. This piece of information is used by the Maude engine that matches the equations in the module regardless of how parentheses are left- or right-associated. Moreover the simpler syntax $\texttt{seq}(t_1, t_2 \ldots, t_n)$ can be used for any $n \in \mathbb{N}$, $n > 1$, exploiting the associativity of `seq`. Similarly, the attribute `id: nilSeq` says that `nilSeq` is the identity for `seq`. In general, the Maude engine can rewrite modulo different combinations of associativity, commutativity, identity (left-, right-, or two sided), and idempotency. Therefore, data structures as lists, sets, and multisets can be naturally represented in Maude.

Now, we have the basis for the definition of a module `TREE`, extending `ND-SEM` by breadth-first and depth-first visit mechanisms for the tree of nondeterministic rewritings in T. A *strategy expression* in `TREE` has either the form $\texttt{rewWith}(\overline{T}, \overline{t}, S)$ where S is the rewriting strategy that one wishes to compute, or the form `failure`, which means that something has gone wrong. As the computation of a given strategy proceeds, t is rewritten in T according to S (and S is reduced into the remaining strategy to be computed). In case of termination, S becomes the trivial strategy `idle`. In what follows, we assume the existence of a user-definable predicate `ok(_)` for expressing success

or failure at the object level, as defined in Section 3.1.

```
mod TREE is protecting ND-SEM .
 sorts TermSet Strategy StrategyExpression .
 subsort Term < TermSet .
 op isIn : Term TermSet -> Bool .
 op emptySet : -> TermSet .
 op set : TermSet TermSet -> TermSet [assoc comm id: emptySet] .
 op idle : -> Strategy . op failure : -> StrategyExpression .
 op rewWith : Module Term Strategy -> StrategyExpression .
 ops breadth depth : Label -> Strategy .
 ops rewWithBF rewWithDF : Module TermSeq TermSet Label ->
                           StrategyExpression .
 var T : Module . vars t t' : Term . var n : Nat .
 var TS : TermSet . vars TL TL' : TermSeq . var l : Label .
 eq set(t,t) = t .
 eq rewWith(T,t,breadth(l)) = rewWithBF(T,t,emptySet,l) .
 eq rewWithBF(T,nilSeq,TS,l) = failure .
 eq rewWithBF(T,seq(t,TL),TS,l) =
    if isIn(t,TS) then rewWithBF(T,TL,TS,l)
    else (if meta-reduce(T,'ok[t]) == 'true
          then rewWith(T,t,idle)
          else (if meta-reduce(T,'ok[t]) == 'false
                then rewWithBF(T,TL,set(t,TS),l)
                else rewWithBF(T,seq(TL,allRew(T,t,l)),
                     set(t,TS),l)
                fi) fi) fi .
```

The expression `rewWith(`T`,`t`,breadth(`l`))` rewrites a term t in T using rules with label l, and exploring all the possibilities "in parallel" (using the breadth-first visit) until a solution is found. The function `rewWithBF` takes as arguments the metarepresentation of a module T, a sequence of metaterms TL, a set of metaterms TS and the representation of a label l. The set TS represents the set of already visited terms. The sequence TL contains the terms that have not been checked yet. If the second argument is the empty sequence, then the function evaluates to `failure` (i.e., no solution is reachable, because all the possible computations fail). If there is at least one element t in the sequence, such that $t \notin TS$ and `ok(`t`) = false`, then all the possible rewritings of t in T via rules with label l are appended to the rest of the list (e.g., the sequence of terms is managed as a queue). If `ok(`t`) = true`, then t is a solution, the evaluation returns `rewWith(`T`,`t`,idle)` and we are done.

The implementation of the strategy `depth(`l`)` for the depth-first visit of the tree is very similar to the previous one (and thus omitted), except that the sequence TL in `rewWithDF(`T`,`TL`,`TS`,`l`)` is managed as a stack instead of as a queue. This solution does not correspond exactly to the classical notion,

because once a term t is selected, all of its possible rewritings are calculated. To improve efficiency, we define the following variant: the stack contains pairs of the form $(\overline{t}, i)$, where t is a term and i is a natural number. When such a pair is selected, it means that only the first $i-1$ rewritings of t have been already inspected and that the i-th rewriting t_i of t (if any) should be examined next. The advantage of this strategy is that the stack remains smaller in size, because each rewriting is computed by need. We use the name `bcktr` for this strategy, because it implements a sort of backtracking mechanism. Since this strategy yields the same result as the `depth` strategy, in what follows we do not specify which one is used when a depth-first visit is involved.

```
sorts Pair PairSeq .
subsort Pair < PairSeq .
op nilPair : -> Pair . op pair : Term Nat -> Pair .
op seqPair : PairSeq PairSeq -> PairSeq [assoc id : nilPair] .
op rewWithBT : Module PairSeq TermSet Label ->
               StrategyExpression .
op bcktr : Label -> Strategy .
var PL : PairSeq .
eq rewWith(T,t,bcktr(l)) = rewWithBT(T,pair(t,0),emptySet,l) .
eq rewWithBT(T,nilSeq,TS,l) = failure .
eq rewWithBT(T,seqPair(pair(t,n),PL),TS,l) =
   if isIn(t,TS) then rewWithBT(T,PL,TS,l)
   else (if meta-reduce(T,'ok[t]) == 'true
         then rewWith(T,t,idle)
         else (if meta-reduce(T,'ok[t]) == 'false
               then rewWithBT(T,PL,set(t,TS),l)
               else (if meta-apply(T,t,l,n) == {error*,empty}
                     then rewWithBT(T,PL,set(t,TS),l)
                     else rewWithBT(T,seqPair(
                          pair(extTerm(meta-apply(T,t,l,n)),0),
                          pair(t,succ(n)),PL),set(t,TS),l)
                     fi) fi) fi) fi .
```

Using these strategies, the solution (if any) is processed in a deterministic way. It is also possible to define a nondeterministic visit mechanism of the tree, where the nondeterminism is due to the choice of the term to be rewritten from the list of already reached terms. Once a term t is selected from the list, there are two possibilities. If t is successful, then we can discharge all the other branches, and t is returned as a solution. If t is not successful then the computation proceeds by exploring also the rewritings of t. If we only select terms that are not successful, then at some point we will reach either an empty list of terms to be checked, or the list containing all the successful states. Notice that there is only one solution of this kind, and that all the computation paths leading to that solution have always the same length. It follows that the meaningful nondeterminism consists in selecting a successful

term from the list, because it can lead to different final states. Since we look for some control mechanism over nondeterministic computations, we could use a rewriting rule (with label `aux`) instead of an equation. The resulting evaluation strategy is: "Recursively expand any term that is not a solution and eventually choose one of the solutions (if any)". It follows that at the meta-meta-level, only one step of meta-rewriting is needed, to find a solution, and that the operation `allRew` can be used in the module `ND-SEM[TREE[`T`]]` to collect all the meta-solutions (that have the form `rewWith(`$\overline{T}$`,`$\bar{t}$`,idle)` with `ok(`t`) = true` in T). We use an auxiliary predicate `okSeq` to recognize the sequences of solutions.

```
op nondet : Label -> Strategy .
op rewWithND : Module TermSeq TermSet Label ->
               StrategyExpression .
op okSeq : Module Term -> Bool .
eq okSeq(T̄,nilSeq) = true .
eq okSeq(T̄,seq(t̄,TL)) =
   if meta-reduce(T̄,'ok[t̄]) == 'true then okSeq(T̄,TL)
   else false fi .
eq rewWith(T̄,t̄,nondet(l̄)) = rewWithND(T̄,t̄,emptySet,l̄) .
eq rewWithND(T̄,nilSeq,TS,l̄) = failure .
ceq rewWithND(T̄,seq(TL,t̄,TL'),TS,l̄) =
    if isIn(t̄,TS) then rewWithND(T̄,seq(TL,TL'),TS,l̄)
    else (if meta-reduce(T̄,'ok[t̄]) == 'false
          then rewWithND(T̄,seq(TL,TL'),set(t̄,TS),l̄)
          else rewWithND(T̄,seq(TL,allRew(t̄,l̄),TL'),
               set(t̄,TS),l̄)
          fi) fi
    if meta-reduce(T̄,'ok[t̄]) =/= 'true .
crl [aux] : rewWithND(T̄,seq(TL,t̄,TL'),TS,l̄) =>
            rewWith(T̄,t̄,idle)
    if okSeq(T̄,seq(TL,t̄,TL')) .
```

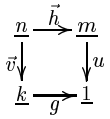
To summarize: given a nondeterministic rewriting specification T (containing the success predicate `ok`), then: (1) the module `ND-SEM[`T`]` allows collecting and analyzing all the possible one-step rewritings of a term; (2) the module `TREE[`T`]` allows analyzing one solution among those reachable from a term, and the chosen solution depends on the adopted strategy; and (3) the module `ND-SEM[TREE[`T`]]` allows collecting and analyzing all the possible (topmost) solutions reachable from a term.

4 Nondeterminism and Uniform Term Tile Systems

Let $\mathcal{R}$ be a tile system, and let $\mathcal{R}'$ denote its translation in rewriting logic (as described in Section 2). The tile system $\mathcal{R}$ is called *uniform* if for each

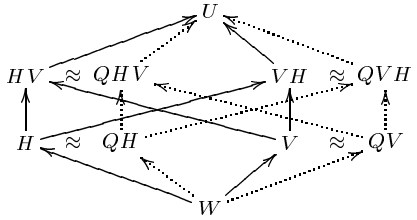
sequent $s; b \Rightarrow a; s'$ entailed in $\mathcal{R}'$ such that s, s' are horizontal arrows and a, b are vertical arrows, then there exists a sequent $s \xrightarrow[b]{a} s'$ entailed by $\mathcal{R}$. It follows that a general notion of success for uniform tile systems consists of VH configurations (in the sense of a Vertical arrow followed by a Horizontal arrow), and the general strategies presented in Section 3 can be immediately applied to compute tile systems. In this section we show how to employ the membership assertions of Maude to model uniform term tile systems (tTS).

If both configurations and effects are terms over two distinct (unsorted) signatures Σ_H and Σ_V, then we can assume a standard representation [6] of basic tiles having the form



(with $\vec{h} \in T_{\Sigma_H}(X_n)^m$, $g \in T_{\Sigma_H}(X_k)$, $\vec{v} \in T_{\Sigma_V}(X_n)^k$, and $u \in T_{\Sigma_V}(X_m)$, where $X_i = \{x_1, ..., x_i\}$ is a chosen set of variables, totally ordered by $x_{j_1} < x_{j_2}$ iff $j_1 < j_2$) as sequents $n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle u \rangle]{\langle \vec{v} \rangle} \langle g \rangle$, where the number of variables in the "upper-left" corner of the tile is made explicit (the values m and k can be retrieved from the lengths of the term vectors decorating the tile). Since sequential composition of terms is just substitution, then the translation of such a tile returns the rule $u[\vec{h}/\vec{x}] \Rightarrow g[\vec{v}/\vec{x}]$, where $t[\vec{w}/\vec{x}]$ denotes the *simultaneous* substitution of each w_i for x_i in t.

Given a uniform tTS $\mathcal{R}$, we can define a rewrite theory $\hat{\mathcal{R}}$ whose equational part in membership equational logic has the poset of sorts illustrated below:



The sort W informally contains the variables of the system as constants; the sort H contains the terms over the signature Σ_H and variables in W (similarly for the sort V); the sort HV contains those terms over the signature $\Sigma_{H \cup V}$ and variables in W such that they are decomposable as terms over the signature Σ_V applied to terms over Σ_H (similarly for VH); sorts QH, QV, QHV, and QVH are *quoted versions* of sorts H, V, HV, and VH (we will denote the quoted version of a signature Σ_S by $\Sigma_{S'}$, assuming that the operators of the latter are the *syntactically* quoted version of the operators in Σ_S). The sort U contains terms over the signature $\Sigma_{H \cup V \cup H' \cup V'}$ and variables in W. As summarized above, we introduce the corresponding operations and membership assertions, for each $h \in \Sigma_H$ and $v \in \Sigma_V$ and their quoted version (with h of arity n and v of arity m). We also add two operations for transforming a term into its quoted version and vice versa, together with the adequate membership assertions.

Then, we introduce an operator `top(_)` to mark the term to be rewritten and two rules to begin and terminate the rewriting computation. The basic rules are the quoted and stretched versions of the tiles in $\mathcal{R}$. Then, Theorem 4.1 may be easily proved via a simple inspection of the rules in $\hat{\mathcal{R}}$.

`ops h qh :` $\mathtt{U}^n$ `-> U . ops v qv :` $\mathtt{U}^m$ `-> U . vars` $x_1 \cdots x_{max}$ `: U .`
`cmb h(`$x_1,\ldots,x_n$`) : H iff` x_1 `...` x_n `: H .`
`cmb v(`$x_1,\ldots,x_m$`) : V iff` x_1 `...` x_m `: V .`
`cmb h(`$x_1,\ldots,x_n$`) : VH iff` x_1 `...` x_n `: VH .`
`cmb v(`$x_1,\ldots,x_m$`) : HV iff` x_1 `...` x_m `: HV .`
`cmb qh(`$x_1,\ldots,x_n$`) : QH iff` x_1 `...` x_n `: QH .`
`cmb qv(`$x_1,\ldots,x_m$`) : QV iff` x_1 `...` x_m `: QV .`
`cmb qh(`$x_1,\ldots,x_n$`) : QVH iff` x_1 `...` x_n `: QVH .`
`cmb qv(`$x_1,\ldots,x_m$`) : QHV iff` x_1 `...` x_m `: QHV .`
`ops quote unquote top : U -> U .`
`cmb quote(`x_1`) : QH iff` x_1 `: H .`
`cmb quote(`x_1`) : QV iff` x_1 `: V .`
`cmb quote(`x_1`) : QHV iff` x_1 `: HV .`
`cmb quote(`x_1`) : QVH iff` x_1 `: VH .`
`cmb quote(`x_1`) : W iff` x_1 `: W .`
`cmb unquote(`x_1`) : H iff` x_1 `: QH .`
`cmb unquote(`x_1`) : V iff` x_1 `: QV .`
`cmb unquote(`x_1`) : HV iff` x_1 `: QHV .`
`cmb unquote(`x_1`) : VH iff` x_1 `: QVH .`
`cmb unquote(`x_1`) : W iff` x_1 `: W .`
`eq quote(h(`$x_1,\ldots,x_n$`)) = qh(quote(`x_n`),...,quote(`x_n`)) .`
`eq quote(v(`$x_1,\ldots,x_m$`)) = qv(quote(`x_1`),...,quote(`x_m`)) .`
`ceq quote(`x_1`) =` x_1 `if` x_1 `: W .`
`eq unquote(qh(`$x_1,\ldots,x_n$`)) = h(unquote(`x_1`),...,unquote(`x_n`)) .`
`eq unquote(qv(`$x_1,\ldots,x_m$`)) = v(unquote(`x_1`),...,unquote(`x_m`)) .`
`ceq unquote(`x_1`) =` x_1 `if` x_1 `: W .`
`crl top(`x_1`) => top(quote(`x_1`)) if` x_1 `: HV .`
`crl top(`x_1`) => top(unquote(`x_1`)) if` x_1 `: QVH .`
`rl qu(`$\overrightarrow{\mathtt{qh}}(x_1,\ldots,x_n)$`) => qv(`$\overrightarrow{\mathtt{qg}}(x_1,\ldots,x_n)$`) .`

Theorem 4.1 ([6]) *Given a uniform tTS $\mathcal{R}$, then*

$$\mathcal{R} \vdash n \triangleleft \langle \vec{h} \rangle \xrightarrow[\langle u \rangle]{\langle \vec{v} \rangle} \langle g \rangle \iff \hat{\mathcal{R}} \vdash \mathit{top}(u(\overrightarrow{h})) \Rightarrow \mathit{top}(g(\overrightarrow{v})).$$

A simple predicate `ok` that uses the membership expressivity of Maude to distinguish `VH` states, gives the right notion of success for uniform tTS and allows the immediate application of the search strategies defined in Section 3.3:

`ceq ok(top(`x_1`)) = true if` x_1 `: VH .`

Remark 4.2 If the tTS is not uniform, then the actual proof term decorating the derivation has also to be taken into account. Consequently, the meta-

strategies need to be changed in order to record not only the state, but also the derivation steps which led to that state. This means that the structure of the meta-state would become huge very fast during the execution, affecting the computations, that would become very slow. Since at present we do not have any meaningful examples of non-uniform systems we are not really interested in having such an implementation.

4.1 Example: Finite CCS

Due to space limitation, we illustrate the application of the internal strategies described in the previous sections only for the simple example of *finite CCS*. We refer the reader to [6] for a more interesting example dealing with the tile system proposed by Ferrari and Montanari in [15] for *located CCS* [4].

Milner's *Calculus for Communicating Systems* (CCS) [27] is among the more well-known and studied concurrency models. We present here an executable tile specification for a fragment of CCS, called *finite CCS*.

Let Δ (ranged over by α) be the set of *basic actions*, and $\bar{\Delta}$ the set of *complementary actions* (with $\Delta = \bar{\bar{\Delta}}$ and $\Delta \cap \bar{\Delta} = \emptyset$). We denote by Λ (ranged over by λ) the set $\Delta \cup \bar{\Delta}$. Let $\tau \notin \Lambda$ be a *distinguished* action, and let $Act = \Lambda \cup \{\tau\}$ (ranged over by μ) be the set of *actions*. Then, a *finite CCS process* is a term generated by the following grammar (including *inactive process*, *action prefix*, *restriction*, *relabelling*, *nondeterministic sum*, and *parallel composition*):

$$P ::= nil \;\mid\; \mu.P \;\mid\; P\backslash\alpha \;\mid\; P[\alpha/\beta] \;\mid\; P + P \;\mid\; P|P.$$

We let P, Q range over the set $Proc$ of processes. A transition system $T_{CCS} \subseteq Proc \times Act \times Proc$ (presented in SOS style) usually describes the operational semantics of CCS. Assuming the reader familiar with the subject, we skip its formal definition. As usual we write $P \xrightarrow{\mu} Q$ instead of $(P, \mu, Q) \in T_{CCS}$, meaning that a process P may perform an action μ becoming Q. We adapt the tile system for CCS proposed in [19] to settle the following tTS.

Definition 4.3 [tTS for finite CCS] The tTS $\mathcal{R}_{CCS}$ has the signature $\Sigma_A = \{\mu : 1 \longrightarrow 1 \mid \mu \in Act\}$ as horizontal signature, the signature Σ_P of CCS processes as vertical signature, and the following basic tiles:

$$act_\mu : 1 \triangleleft \langle x_1 \rangle \xrightarrow[\langle \mu.x_1 \rangle]{\langle x_1 \rangle} \langle \mu(x_1) \rangle \qquad \|_\lambda : 2 \triangleleft \langle \lambda(x_1), \bar{\lambda}(x_2) \rangle \xrightarrow[\langle x_1 | x_2 \rangle]{\langle x_1 | x_2 \rangle} \langle \tau(x_1) \rangle$$

$$\langle +_\mu : 2 \triangleleft \langle \mu(x_1), x_2 \rangle \xrightarrow[\langle x_1 + x_2 \rangle]{\langle x_1 \rangle} \langle \mu(x_1) \rangle \qquad \rfloor_\mu : 2 \triangleleft \langle \mu(x_1), x_2 \rangle \xrightarrow[\langle x_1 | x_2 \rangle]{\langle x_1 | x_2 \rangle} \langle \mu(x_1) \rangle$$

$$+\rangle_\mu : 2 \triangleleft \langle x_1, \mu(x_2) \rangle \xrightarrow[\langle x_1 + x_2 \rangle]{\langle x_2 \rangle} \langle \mu(x_1) \rangle \qquad \lfloor_\mu : 2 \triangleleft \langle x_1, \mu(x_2) \rangle \xrightarrow[\langle x_1 | x_2 \rangle]{\langle x_1 | x_2 \rangle} \langle \mu(x_1) \rangle$$

$$res_{\mu,\alpha} : 1 \triangleleft \langle \mu(x_1) \rangle \xrightarrow[\langle x_1 \backslash \alpha \rangle]{\langle x_1 \backslash \alpha \rangle} \langle \mu(x_1) \rangle \;\; (\text{if } \mu \notin \{\alpha, \bar{\alpha}\})$$

$$rel_{\mu,\alpha,\beta} : 1 \triangleleft \langle \mu(x_1) \rangle \xrightarrow[\langle x_1[\alpha/\beta] \rangle]{\langle x_1[\alpha/\beta] \rangle} \langle t \rangle \text{ with } t = \begin{cases} \beta(x_1) \text{ if } \mu = \alpha \\ \bar{\beta}(x_1) \text{ if } \mu = \bar{\alpha} \\ \mu(x_1) \text{ otherwise} \end{cases}$$

Here, the vertical dimension is associated with process descriptions, and the horizontal dimension represents the (*opposite* of the) dynamic evolution of the system (we say *opposite*, because the arrows representing the actions performed by the system are reversed from their computational-driven intuitive direction). For the reader already acquainted with the Gadducci and Montanari's tile system, the previous definition may appear somewhat confusing, because the two dimensions are swapped in a counterintuitive way. The reason is that the direct translation of our system in a Maude module allows collecting the possible evolutions of an agent, whereas the ordinary definition would allow only the test of executable actions (as explained in Section 2). Analogously to [19], the following result holds, establishing the correspondence from the set-theoretic view of the traditional SOS semantics for CCS, and the sequents entailed by $\mathcal{R}_{CCS}$.

Theorem 4.4 *The tTS $\mathcal{R}_{CCS}$ is uniform, and for any CCS agents P, Q and for any action μ:*

$$P \xrightarrow{\mu} Q \iff \mathcal{R}_{CCS} \vdash 0 \triangleleft \langle\rangle \xrightarrow[\langle Q \rangle]{\langle P \rangle} \langle \mu(x_1) \rangle \,.$$

It follows that a suitable implementation of $\mathcal{R}_{CCS}$ can be obtained by considering the rewriting system $\hat{\mathcal{R}}_{CCS}$ as defined in Section 4. We sketch the Maude description of the module CCS.

```
mod CCS is sorts W H V VH HV U QH QV QVH QHV .
 subsorts W <  H V  <  VH HV  < U .
 subsorts W < QH QV < QVH QHV < U .
 sorts Channel Act . subsort Channel < Act .
 ops quote unquote top : U -> U .
 op a : Nat -> Channel . op tau : -> Act .
 op bar : Channel -> Channel .
 ops nil qnil : -> U . ops pre qpre : Act U -> U .
 ops res qres : U Channel -> U .
 ops rel qrel : U Channel Channel -> U .
 ops plus par qplus qpar : U U -> U .
 ops obs qobs : U Act -> U .
 vars P Q : U . var A : Act . vars C D : Channel .
 eq bar(bar(C)) = C .

 mb nil : V .
 cmb pre(A,P) : V if P : V .
 cmb res(P,C) : V if P : V .
```

```
cmb rel(P,C, D) : V if P : V .
cmb plus(P,Q) : V if P : V and Q : V .
cmb par(P,Q) : V if P : V and Q : V .
cmb obs(P,A) : H if P : H .
mb nil : HV .
cmb pre(A,P) : HV if P : HV .
cmb res(P,C) : HV if P : HV .
cmb rel(P,C,D) : HV if P : HV .
cmb plus(P,Q) : HV if P : HV and Q : HV .
cmb par(P,Q) : HV if P : HV and Q : HV .
cmb obs(P,A) : VH if P : VH .
mb qnil : QV .
cmb qpre(A,P) : QV if P : QV ...
*** we omit the rest of similar membership axioms
*** relative to the quoted operators

cmb quote(P) : QH if P : H .
cmb quote(P) : QV if P : V .
cmb quote(P) : QVH if P : VH .
cmb quote(P) : QHV if P : HV .
cmb quote(P) : W if P : W .
eq quote(nil) = qnil .
eq quote(pre(A,P)) = qpre(A,quote(P)) .
eq quote(res(P,C)) = qres(quote(P),C) .
eq quote(rel(P,C,D)) = qrel(quote(P),C,D) .
eq quote(plus(P,Q)) = qplus(quote(P),quote(Q)) .
eq quote(par(P,Q)) = qpar(quote(P),quote(Q)) .
eq quote(obs(P,A)) = qobs(quote(P),A) .
ceq quote(P) = P if P : W .
cmb unquote(P) : H if P : QH .
cmb unquote(P) : V if P : QV ...
*** we omit the rest of similar axioms and equations
*** relative to the operator 'unquote'

crl [qr] : top(P) => top(quote(P)) if P : HV .
crl [qr] : top(P) => top(unquote(P)) if P : QVH .
rl  [qr] : qpre(A,P) => qobs(P,A) .
crl [qr] : qres(qobs(P,A),C) => qobs(qres(P,C),A)
    if A =/= C and A =/= bar(C) .
crl [qr] : qrel(qobs(P,A),C,D) => qobs(qrel(P,C,D),A)
    if A =/= C and A =/= bar(C) .
crl [qr] : qrel(qobs(P,A),C,D) => qobs(qrel(P,C,D),D)
    if A == C .
crl [qr] : qrel(qobs(P,A),C,D) => qobs(qrel(P,C,D),bar(D))
```

```
     if A == bar(C) .
 rl  [qr] : qplus(qobs(P,A),Q) => qobs(P,A) .
 rl  [qr] : qplus(Q,qobs(P,A)) => qobs(P,A) .
 rl  [qr] : qpar(qobs(P,A),Q) => qobs(qpar(P,Q),A) .
 rl  [qr] : qpar(Q,qobs(P,A)) => qobs(qpar(Q,P),A) .
 crl [qr] : qpar(qobs(P,C),qobs(Q,D)) => qobs(qpar(P,Q),tau)
     if C == bar(D) .
 op ok : U -> Bool .
 ceq ok(top(P)) = true if P : VH .
endm
```

The code exactly corresponds to the translation illustrated in section 4, but we use a more verbose syntax for the operators of the tTS. In particular, we assume that: the denumerable set of basic actions is $\{\mathtt{a}(i) \mid i \in \mathbb{N}\}$, the special action τ is denoted by `tau`, the inactive process nil is denoted by `nil`, the action prefix $\mu.P$ is denoted by `pre(`μ`,`P`)`, the restriction $P\backslash\alpha$ is denoted by `res(`P`,`α`)` the relabelling $P[\alpha/\beta]$ is denoted by `rel(`P`,`α`,`β`)`, the nondeterministic sum $P + Q$ is denoted by `plus(`P`,`Q`)`, the parallel composition $P|Q$ is denoted by `par(`P`,`Q`)`, and the dynamic evolution $\mu(P)$ is denoted by `obs(`P`,`μ`)`. Notice that the predicate `ok` is exactly the one illustrated in Section 4 for a generic uniform tTS.

Remark 4.5 The sort `W` is necessary for executing partially specified queries (in this case the process variables that are used must be declared as constants having sort `W`).

Example 4.6 We show the result of a computation in `ND-SEM[TREE[CCS]]`, collecting the successful evolutions of the process $(a_1.nil + a_2.nil)|\bar{a}_1.nil$. The meta-meta-notation could require some acquaintance with meta-translations, but some tools will soon be available to perform automatic translations. Notice that all the possible interleaving computations of the initial process are collected (the last answer corresponds to the idle computation).

```
Maude> rew allRew('rewWith['_[_]['']top,'_[_]['']par,'_,_['_[_][
       ''plus,'_,_['_[_]['']pre,'_,_['_[_]['']a,''1],''nil]],
       '_[_]['']pre,'_,_['_[_]['']a,''2],''nil]]]],
       '_[_]['']pre,'_,_['_[_]['']bar,'_[_]['']a, ''1]],''nil]]]]],
       'nondet['']qr]],'aux) .

rewrites: 26822 in 719ms cpu (729ms real)(37252 rewrites/second)
result TermSequence: seq(
*** a1(nil|ā1.nil)
'rewWith[('_[_]['']top,('_[_]['']obs,('_,_[('_[_]['']par,('_,_[
''nil,('_[_]['']pre,('_,_[('_[_]['']bar,('_[_]['']a,''1])]),''nil])
])])]),('_[_]['']a,''1])])])]),'idle],
*** a1(ā1(nil|nil))
'rewWith[('_[_]['']top,('_[_]['']obs,('_,_[('_[_]['']obs,('_,_[
```

```
('_[_]['']par,('_,_['']nil,''nil])]),('_[_]['']bar,('_[_]['']a,''1])
])])]),('_[_]['']a,''1])])])]),'idle],
```

*** $\bar{a}_1(a_1(nil|nil))$

```
'rewWith[('_[_]['']top,('_[_]['']obs,('_,_[('_[_]['']obs,('_,_[
('_[_]['']par,('_,_['']nil,''nil])]),('_[_]['']a,''1])])]),
('_[_]['']bar,('_[_]['']a,''1])])])])]),'idle],
```

*** $\tau(nil|nil)$

```
'rewWith[('_[_]['']top,('_[_]['']obs,('_,_[('_[_]['']par,('_,_[
''nil,''nil])]),''tau])])]),'idle],
```

*** $a_2(nil|\bar{a}_1.nil)$

```
'rewWith[('_[_]['']top,('_[_]['']obs,('_,_[('_[_]['']par,('_,_[
''nil,('_[_]['']pre,('_,_[('_[_]['']bar,('_[_]['']a,''1])]),''nil])
])])]),('_[_]['']a,''2])])])]),'idle],
```

*** $a_2(\bar{a}_1(nil|nil))$

```
'rewWith[('_[_]['']top,('_[_]['']obs,('_,_[('_[_]['']obs,('_,_[
('_[_]['']par,('_,_['']nil,''nil])]),('_[_]['']bar,('_[_]['']a,''1])
])])]),('_[_]['']a,''2])])])]),'idle],
```

*** $\bar{a}_1(a_2(nil|nil))$

```
'rewWith[('_[_]['']top,('_[_]['']obs,('_,_[('_[_]['']obs,('_,_[
('_[_]['']par,('_,_['']nil,''nil])]),('_[_]['']a,''2])])]),('_[_][
''bar,('_[_]['']a,''1])])])])]),'idle],
```

*** $\bar{a}_1((a_1.nil + a_2.nil)|nil)$

```
'rewWith[('_[_]['']top,('_[_]['']obs,('_,_[('_[_]['']par,('_,_[
('_[_]['']plus,('_,_[('_[_]['']pre,('_,_[('_[_]['']a,''1]),''nil])
]),('_[_]['']pre,('_,_[('_[_]['']a,''2]),''nil])])])])]),''nil])]),
('_[_]['']bar,('_[_]['']a,''1])])])])]),'idle],
```

*** $(a_1.nil + a_2.nil)|\bar{a}_1.nil$

```
'rewWith[('_[_]['']top,('_[_]['']par,('_,_[('_[_]['']plus,('_,_[
('_[_]['']pre,('_,_[('_[_]['']a,''1]),''nil])]),('_[_]['']pre,(
'_,_[('_[_]['']a,''2]),''nil])])])]),('_[_]['']pre,('_,_[('_[_][
''bar,('_[_]['']a,''1])]),''nil])])])])]),'idle])
```

5 Concluding Remarks

We have implemented and experimented in Maude with the translation of uniform tile systems, using internal strategies to control the intrinsic nondeterminism of the specification. The implementation of finite CCS is extensively discussed and an example illustrates how to compute tiles at the meta-meta-level of the specification. Our experiments are encouraging, since Maude seems to offer a good trade-off between rewriting kernel efficiency and layer swapping management (from terms to their meta-representations and viceversa).

Acknowledgements

We would like to thank Narciso Martí-Oliet for his precious suggestions and comments on a preliminary version of this paper and Manuel Clavel for his help in the area of strategies. We also thank the anonymous referees for their careful reading and their helpful comments.

References

[1] A. Bastiani and C. Ehresmann. Multiple Functors I: Limits Relative to Double Categories, *Cahiers de Top. ed Géo. Diff.* **15**:3, 545–621 (1974).

[2] P. Borovanský, C. Kirchner, and H. Kirchner. Controlling Rewriting by Rewriting In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications. ENTCS* **4** (1996).

[3] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications. ENTCS* **4** (1996).

[4] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing Localities. *TCS* **114**, 31–61 (1993).

[5] R. Bruni. PhD Thesis, Department of Computer Science, University of Pisa, forthcoming.

[6] R. Bruni, J. Meseguer, and U. Montanari. Process and Term Tile Logic. Technical Report SRI-CSL-98-06, SRI International (July 1998). Also Technical Report TR-98-09, Department of Computer Science, University of Pisa (1998).

[7] M.G. Clavel. Reflection in General Logics and in Rewriting Logic with Applications to the Maude Language. PhD Thesis. Univ. de Navarra (1998).

[8] M.G. Clavel, F. Duran, S. Eker, P. Lincoln, and J. Meseguer. An Introduction to Maude (Beta Version). SRI International (March 1998).

[9] M.G. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications. ENTCS* **4** (1996).

[10] M.G. Clavel and J. Meseguer. Reflection and Strategies in Rewriting Logic. In: J. Meseguer, Ed., *Proc. First International Workshop on Rewriting Logic and its Applications. ENTCS* **4** (1996).

[11] M.G. Clavel and J. Meseguer. Axiomatizing Reflective Logics and Languages. In: G. Kiczales, Ed., *Proc. Reflection'96*. San Francisco, USA, 263–288 (1996).

[12] M.G. Clavel and J. Meseguer. Internal Strategies in a Reflective Logic. In: B. Gramlich, and H. Kirchner, Eds., *Proc. of the CADE-14 Workshop on Strategies in Automated Deduction*, Townsville, Australia, 1–12 (1997).

[13] A. Corradini and F. Gadducci. A 2-categorical Presentation of Term Graph Rewriting. In: E. Moggi, G. Rosolini, Eds., *Proc. CTCS'97. LNCS* **1290**, 87–105 (1997).

[14] C. Ehresmann. Catégories Structurées: *I and II, Ann. Éc. Norm. Sup.* **80**, 349–426, Paris (1963); *III, Topo. et Géo. Diff.* **V**, Paris (1963).

[15] G.L. Ferrari and U. Montanari. Tiles for Concurrent and Located Calculi. In: C. Palamidessi, J. Parrow, Eds., *Proceedings of EXPRESS'97. ENTCS* **7** (1997).

[16] K. Futatsugi and T. Sawada. Cafe as an extensible specification environment. In *Proc. of the Kunming International CASE Symp.*, Kunming, China (1994).

[17] F. Gadducci. On the Algebraic Approach to Concurrent Term Rewriting. PhD Thesis TD-96-02. Department of Computer Science, University of Pisa (1996).

[18] F. Gadducci and U. Montanari. Enriched Categories as Models of Computations. In: *Proc. 5th Italian Conference on Theoretical Computer Science, ITCS'95*. World Scientific, 1–24 (1995).

[19] F. Gadducci and U. Montanari. The Tile Model. In: G. Plotkin, C. Stirling, M. Tofte, Eds., *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, to appear. Also Technical Report TR-96-27, Department of Computer Science, University of Pisa (1996).

[20] G.M. Kelly and R.H. Street. Review of the Elements of 2-categories. *Lecture Notes in Mathematics* **420**, 75–103 (1974).

[21] F.W. Lawvere. Functorial Semantics of Algebraic Theories. In *Proc. National Academy of Science* **50**, 869–872 (1963).

[22] S. MacLane. Categories for the Working Mathematician. Springer-Verlag (1971).

[23] N. Martí-Oliet and J. Meseguer. Rewriting Logic as a Logical and Semantic Framework. SRI Technical Report, CSL-93-05 (August 1993). To appear in D. Gabbay, Ed., *Handbook of Philosophical Logic*, Kluwer Academic Publishers.

[24] J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science* **96**, 73–155 (1992).

[25] J. Meseguer. Membership Equational Logic as a Logical Framework for Equational Specification. In: F. Parisi-Presicce, Ed., *Proc. 12th WADT Workshop on Algebraic Development Techniques. LNCS* **1376**, 18–61 (1998).

[26] J. Meseguer and U. Montanari. Mapping Tile Logic into Rewriting Logic. In: F. Parisi-Presicce, Ed., *Proc. 12th WADT Workshop on Algebraic Development Techniques*, *LNCS* **1376**, 62–91 (1998).

[27] R. Milner. Communication and Concurrency. Prentice-Hall (1989).

[28] P. Viry. Rewriting Modulo a Rewrite System. Technical Report TR-95-20. Department of Computer Science, University of Pisa (1995).